Coupling Evaluators for Attribute Coupled Grammars*

Gilles Roussel, Didier Parigot & Martin Jourdan

INRIA
Projet "ChLoE", Bât. 13, Domaine de Voluceau, Rocquencourt, BP 105
F-78153 Le Chesnay Cedex, France
E-mail: {Gilles.Roussel,Didier.Parigot,Martin.Jourdan}@inria.fr

Abstract. Some years ago, the notion of attribute coupled grammars was introduced by Ganzinger and Giegerich [4], together with descriptional composition. The latter works essentially at the specification level, i.e., it produces an attribute grammar which specifies the composition of two attribute coupled grammars.
We introduce a new approach to this composition of attribute coupled grammars. This composition no longer works at the specification level but at the evaluator level. It produces a special kind of attribute evaluator. For this purpose we have introduced the notion of coupling evaluator.
The main advantage of this new approach, compared with descriptional composition, is that it is possible to build separately the coupling evaluator of each attribute coupled grammar; in other words it allows real separate compilation of AG modules.
Another important advantage is that we do not need to check the attribute grammar class in order to construct the final sequence of evaluators; thus, this construction produces a new sort of evaluator.

1 Introduction

Since Knuth's seminal paper introducing attribute grammars (AGs) [10], it has been widely recognized that this method is quite attractive for specifying every kind of syntax-directed computation, the most obvious application being compiler construction. Apart from pure specification-level features —declarativeness, structure, locality of reference— an important advantage of AGs is that they are executable, i.e., it is possible to automatically construct, from an AG specifying some computation, a program which implements it. One could thus expect that they are used heavily to develop practical, production-quality, applications.

Unfortunately, it appears that this is not yet the case, although AGs have been around for quite a long time and although powerful AG-processing systems are now available (e.g., Fnc-2 [9], which is the base of the present work; see also [1] for a good list of other systems). In our opinion, the main reason for this is that AGs still cruelly lack the same support for modularity as the one which is offered by most programming languages, even the oldest ones [8].

* This work was supported in part by ESPRIT Project #5399 "Compare".

This is the reason why Attribute Coupled Grammars and Attribute Couplings (AC) were introduced in [4] to allow modularity in AG specifications. An application can be decomposed into several attribute couplings, each of which transforms an input syntax tree into a new output syntax tree. This has been widely recognized as a very important concept for attribute grammar modularization.

As separate compilation of each AC into a classical evaluator leads to a loss of efficiency compared to non-modular specifications (because of intermediate tree constructions), descriptional composition has been introduced to create, from a modular specification, a large attribute grammar which avoids any intermediate tree construction.

In our system based on Strongly Non-Circular (SNC) attribute grammars [9], this construction looses a bit of power because the SNC class is not closed under descriptional composition [6]: the attribute grammar resulting from the descriptional composition of SNC modules is not necessarily SNC. Moreover, in the context of attribute grammar reuse [3], descriptional composition appeared to be limited due to the need for complete reconstruction of the resulting attribute grammar before any evaluator construction: separate compilation is impossible.

These observations and our personal work on attribute grammars reusability [7,11] have led us to introduce a new technique allowing separate compilation of modules as in [3] but with no intermediate tree construction. This technique is based on the simple but powerful idea of descriptional composition, i.e. attaching computations of an attribute grammar to the tree construction actions of the attribute coupling which precedes it in a sequence. The main difference between descriptional composition and our technique is that we do not work at the specification level but at the evaluator level. To do so, we introduce new sorts of evaluators, called *coupling evaluators* and *parameterizable evaluators*.

Given a sequence of ACs, we construct a parameterizable evaluator for the last AC. This evaluator is very similar to a classical evaluator, except that it only sees one special production instance of its input tree at each time. When placed at the end of a sequence, when it wants to move in its input tree (change to a neighbouring node, i.e. another production instance), it queries the "visit selector" mechanism which performs the move in the input tree of the first grammar in the sequence. The result is a new production instance which is passed through the coupling evaluators attached to the different ACs of the sequence. Each of them transforms its input production instance into a new production instance for its output grammar. The last created production instance is the production instance that the parameterizable evaluator requires.

Beside good properties of separate compilation and efficiency, this technique brings an interesting result for attribute grammar evaluators: since it avoids all problems of class closure under descriptional compilation, it can construct an evaluator based on visit sequences for a non-SNC attribute grammar, provided that it is correctly modularized [6]. This method essentially differs from the approach in [3], by providing separate compilation while keeping the property of the descriptional composition that the intermediate trees are not constructed.

2 Outline of Attribute Couplings and their Descriptional Composition

First, we recall some notations and definitions on Attribute Grammars and Attribute Couplings, but for complete definitions readers should refer to the excellent paper by Ganzinger and Giegerich [4].

Definition 1. A *context-free grammar* is a tuple $G = (N, T, Z, P)$ in which:

- N is a set of non-terminals;
- T is a set of terminals, $N \cap T = \emptyset$;
- Z is the root non-terminal (start symbol), $Z \in N$;
- P is a set of productions,
  $p : X_0 \rightarrow X_1 \ldots X_n$ with $X_0 \in N$ and $X_i \in (T \cup N)$.

Notations for a grammar G:

t denotes a syntax tree of G;
u denotes a node of t;
$label(u)$ notes the non-terminal at u;
$prod(u)$ denotes the production at u;
$Pos(u, i)$ denotes the child of node u at position i.

Other notations will be defined as needed, but most of them are quite well-known or hopefully self-explanatory. To introduce our coupling evaluators, we first recall the definition of attribute grammars where the attributes are typed by sorts.

Definition 2 (Attribute Grammar). An *Attribute Grammar* is a tuple $AG = (G, S, A, F)$ where :

- $G = (N, T, Z, P)$ is a context-free grammar as in definition 1.
- S is a set of sorts;
- $A = \bigcup A(X)$ is a set of attributes attached to $X \in N$, noted $X.a$, with $type(a) \in S$;
- $F = \bigcup F(p)$ is a set of attribute rules where f_{p,a,X_i} designates the attribute rule defining the attribute a of non-terminal X_i in production p.

Non-terminals and terminals of a given grammar can be sorts for some attribute grammar, whose result is then one or more trees of this grammar. Productions are constructors for non-terminal sorts, and terminal sorts are identified with the single constant they contain.

We introduce a simple definition of Attribute Couplings, derived from the one by Ganzinger and Giegerich [4]. Our definition is a restriction of the latter; it disallows any semantic computation and forbids the attribute rules of F to be complex tree-construction rules. We say that this kind of AC is a *purely syntactic AC*. With these restrictions, the definition of descriptional composition and our construction of coupling evaluators are simpler. At the end of this paper we give an informal presentation of how to relax these restrictions.

Definition 3 (Attribute Coupling). We call *Attribute Coupling* of G_1 and G_2, noted $\alpha : G_1 \rightarrow G_2$, an attribute grammar $\alpha = (G_1, S_\alpha, A_\alpha, F_\alpha)$ such that:

- $S_\alpha \subset (N_2 \cup T_2)$;
- each $f_{p,a,X_i} \in F_\alpha$ is either a copy rule or a tree-construction rule associated with a production of P_2;
- the root non-terminal Z_1 has a unique attribute z_2 of type Z_2.

An AC $\alpha : G_1 \rightarrow G_2$ takes as input a tree of G_1 and gives as output a tree of G_2.

For descriptional composition to be well defined, we require that each attribute occurrence of a given production appears once and only once in the attribute rules, either in the LHS (if it is defined) or in the RHS (if it is used). Each attribute occurrence of a given production can thus be associated with a unique attribute rule in this production.

In the sequel, for the sake of simplicity, we consider the copy rules, of the form $X.a := Y.b$, as tree-construction rules $X.a := Id_{type(a)}(Y.b)$. So we assume that each attribute coupling is extended with an identity construction Id_X for each non-terminal X in the input grammar. The attribute rules attached to these new productions are only composed of copy rules between the same attributes of the left hand side and the right hand side non-terminal. Thus, in the following definitions, we do not need to add a special case for copy rules.

Example of attribute coupling We have rephrased the example of the list inversion AG [5] using our notation, see figure 1. The input and the output grammar of this AG are the same list grammar. In the next section, we also use this example to construct a coupling evaluator. $\diamond$

$type(z) = Z;$
$type(s) = type(h) = L;$

$p_1 : Z \rightarrow L$	$L.h := p_3();$	$Z.z := p_1(L.s);$
$p_2 : L_0 \rightarrow L_1 D$	$L_1.h := p_2(L_0.h, D);$	$L_0.s := Id_L(L_1.s);$
$p_3 : L \rightarrow$	$L.s := Id_L(L.h);$	

Added production for copy rules:

$$Id_L : L_0 \rightarrow L_1 \; L_0.s := Id_L(L_1.s); \; L_1.h := Id_L(L_0.h);$$

Fig. 1. Attribute coupling performing list inversion

The aim of descriptional composition is to construct, for a given sequence of two attribute couplings, a new attribute coupling which has the same semantics

as this AC sequence. This new AC has the advantage of not performing the construction of the intermediate tree of G_2.

The basic idea of the descriptional composition algorithm is, given two ACs $\alpha : G_1 \rightarrow G_2$ and $\beta : G_2 \rightarrow G_3$, to project the attribute rules of a given production p_2 of G_2 onto the productions of G_1 where there exists an attribute rule performing the construction of p_2. This projection follows the structure of this attribute rule. Descriptional composition is a purely syntactic construction. It does not take into account the semantics of the projected attribute rules.

Moreover the descriptional composition creates a new set of attributes. The attribute names are composed of an attribute name of the first grammar followed by an attribute of the second one. In our construction of coupling evaluators we find the same notion of attribute names (although the attributes are not really declared).

Definition 4 (Descriptional Composition). [2] Let $\alpha : G_1 \rightarrow G_2$ and $\beta : G_2 \rightarrow G_3$ be attribute couplings. The *Descriptional Composition* of α and β generates an attribute coupling $\gamma : G_1 \rightarrow G_3 = (G_1, S_\gamma, A_\gamma, F_\gamma)$ such that:

1. $S_\gamma = S_\beta$;
2. for each attribute $a_\alpha \in A_\alpha(X^1)$ with $type(a_\alpha) = X^2$ and for each attribute $a_\beta \in A_\beta(X^2)$ the attribute $a_\alpha.a_\beta$ is declared in $A_\gamma(X^1)$ with $type(a_\alpha.a_\beta) = type(a_\beta)$;
3. for each production $p^1 : X_0 \rightarrow X_1 \dots X_n \in P_1$ and for each attribute rule $f_{p^1,x^0,X^0} \in F_\alpha(p^1)$ with form $X^0.x^0 = p^2(X^1.x^1, \dots, X^l.x^l)$, and for each attribute rule $f_{p^2,y^0,Y^0} \in F_\beta(p^2)$ with form $Y^0.y^0 = f(Y^1.y^1, \dots, Y^m.y^m)$ on the production p^2 with the form $p^2 : Y_0 \rightarrow Y_1 \dots Y_l$, we define the attribute rule f_{p^1,t^0,T^0} in $F_\gamma(p^1)$ with the form $T^0.t^0 = f(T^1.t^1, \dots, T^m.t^m)$ such that, for each $j \in [0..m]$, $T^j = X_i$ and $t^j = x^i.y^j$ if $Y^j = Y_k$ and $X^k = X_i$ with $k \in [0..l]$ and $i \in [0..n]$.

We do not consider terminals in the previous definitions because their behavior is the same as classical attribute occurrences if we assume them to have a unique attribute with an empty name and the same type as the terminal itself, see [4].

The AG resulting from the descriptional composition of the list inversion AG of Fig. 1 with itself is presented in Fig. 2. As expected, it constructs a copy of the input list, albeit in a rather complicated manner.

3 Coupling Evaluators

The basic idea of coupling evaluators is very similar to that of descriptional composition, except that we work directly on the evaluators and not on the AG specifications. The basic goal for the coupling evaluator for $\alpha : G_1 \rightarrow G_2$ is to

[2] Please take care that X^i and X_i do not represent the same non-terminal in the various definitions.

$$
\begin{array}{lll}
p_1 : Z \rightarrow L & L.h.s := Id_L(L.h.h); & \\
 & Z.z.z := p_1(L.s.s); & L.s.h := p_3(); \\
 & & \\
p_2 : L_0 \rightarrow L_1 D & L_0.h.h := p_2(L_1.h.h, D); & L_1.h.s := Id_L(L_0.h.s); \\
 & L_0.s.s := Id_L(L_1.s.s); & L_1.h.s := Id_L(L_0.h.s); \\
 & & \\
p_3 : L \rightarrow & L.s.s := Id_L(L.h.s); & L.h.h := Id_L(L.s.h);
\end{array}
$$

Fig. 2. Descriptional composition of the list inversion AG with itself

allow to run the evaluator of $\beta : G_2 \rightarrow G_3$ directly on an input syntax tree of G_1. The coupling evaluator transforms an instance of a production of G_1 into the relevant production instance for the evaluator of β, according to the attribute rules of α. We call this constructed production instance, a *virtual production instance.* As these coupling evaluators are to be used in any context of attribute coupling sequences, we introduce the formal definition of the latter.

Definition 5 (Attribute Coupling Sequence). An *Attribute Coupling Sequence*, noted $\Sigma_i^j = G^i \overset{AG^i}{\rightarrow} G^{i+1} \cdots G^j \overset{AG^j}{\rightarrow} G^{j+1}$ is a sequence of attribute couplings AG^k such that $AG^k : G^k \rightarrow G^{k+1}$ for $k \in [i..j]$.

Given an AC sequence Σ, our goal is to build a coupling evaluator for each AC of Σ except the last one. For the last one, we introduce the notion of parameterizable evaluator (see section 4) which is able to work with the result of the sequence of coupling evaluators, i.e., not only the current node, but a virtual production instance. A coupling evaluator transforms a virtual production instance according to tree-construction rules of its associated AC specification. A parameterizable evaluator accesses its "virtual" syntax tree through constructed virtual production instances which are passed as parameters. Thus the intermediate syntax trees are never physically constructed. See the end of section 5 for a discussion about this special last evaluator and separate compilation.

The following definitions of virtual objects (nodes, production instances, etc.) are parametrized by a given AC sequence Σ, for the sake of correctness ("type checking"). These virtual objects are then used to define and construct the coupling evaluators. We'll see however that the coupling evaluator of a given AC is independent from the AC sequence it is embedded in.

A virtual production instance is an object that the coupling evaluator can consider as a production instance of a real tree of its input grammar. Virtual production instances are composed of complex objects, called *virtual nodes.*

A virtual node is associated with a real node of the input tree and with an attribute sequence composed by attributes of each AC in the preceding subsequence, i.e. the calling context of this coupling evaluator. This attribute sequence accounts for the sequence of transformations of virtual productions which are performed through the preceding sequence of coupling evaluators.

Definition 6 (Virtual node). Given a sequence Σ_1^i, we define a *Virtual Node*, noted v, as a sequence $v = u.a^1.\cdots.a^{i-1}$, and its label, noted $label(v)$, such that:

1. if $i = 1$ then $v = u$ and $label(v) = label(u)$ with:
   - u a node for G^1;
   - $label(u) \in N^1 \cup T^1$;
2. if $i > 1$ then $v = v'.a^{i-1}$ and $label(v) = type(a^{i-1})$ with
   - v' is a virtual node for Σ_1^{i-1};
   - a^{i-1} is an attribute of $label(v')$ in AG^{i-1}.

For a given node in the input syntax tree, there exists more than one virtual node. In the virtual nodes, the node part indicates the real node of input syntax tree, and the attribute part specifies to which attribute it refers according to descriptional composition. In fact, if we draw a parallel with descriptional composition, then the attribute part is exactly the name of an attribute in the attribute grammar resulting from descriptional composition of all ACs in the sequence Σ except the last one.

Then the size of the data structure needed is exactly the size of the data structure used in the AG resulting from the descriptional composition. Moreover the virtual nodes are just the input nodes renamed; we insist on the fact that they do not exist physically.

Definition 7 Virtual Production Instance. Given a sequence Σ_1^i and production $p : X_0 \rightarrow X_1 \dots X_n$ of G^i, a *Virtual Production Instance* of p is of the form $V(p) : v_0 \rightarrow v_1 \dots v_n$, such that, for each $k \in [0..n]$:

1. v_k is a virtual node for the sequence Σ_1^i;
2. $label(v_k) = X_k$.

Now we explain the mechanism, called *virtual construction*, which transforms a virtual production instance of an input syntax tree into a virtual production instance of an output syntax tree according to an attribute occurrence of the input production. This transformation derives from the tree-construction rule in which the attribute occurrence appears; it instantiates this tree-construction on the input virtual production instance, and this gives the output instance production. This transformation also records the position of the attribute occurrence instantiation in the output virtual instance production for further transformations.

Definition 8 (Virtual Construction). Let $AG : G \rightarrow G'$ be a attribute coupling in a sequence Σ. We define a *Virtual Construction* on $p : X_0 \rightarrow X_1 \dots X_n$ according to an attribute a, noted $< V(p), j > \underset{a}{\Longrightarrow} < V(p'), j' >$. It takes as input a virtual instance production $V(p) : v_0 \rightarrow v_1 \dots v_n$ of p and a position j in p. It returns the pair $< V(p'), j' >$ such that:

1. Let $f_{p,y^0,Y^0} \in F$ be the unique attribute rule referencing attribute occurrence $X_j.a$ in p; it is of the form $Y^0.y^0 = p'(Y^1.y^1, \dots, Y^l.y^l)$, with p' a production of G' and $Y^{j'}.y^{j'} = X_j.a$;

2. $V(p') : w_0 \rightarrow w_1 \ldots w_l$ where $w_s = v_k.y^s$ if $Y^s = X_k$ for $s \in [0..l]$.

As we have qssumed that each attribute occurrence was associated with a unique attribute rule, this virtual construction is well defined.

The virtual construction only depends on AG, not on its context in the sequence Σ.

We introduce another notion, the *virtual attribute*, to complete our construction of a coupling evaluator. A virtual attribute is a sequence of attributes composed of one attribute of each AC in the sequence. It is used to direct the sequence of virtual constructions. We will see later that such a virtual attribute represents the name of the calling virtual node in the parameterizable evaluator of the last AC of the sequence.

Definition 9 (Virtual Attribute). Given a sequence Σ_j^m, we define a *Virtual Attribute*, noted $v(a^j)$, as a sequence of attributes $v(a^j) = a^j.\cdots.a^{m-1}$, such that:

1. if $j = m$ then $v(\epsilon) = \epsilon$ (empty sequence[3]) is the only possible virtual attribute;
2. if $j < m$ then $v(a^j) = a^j.v(a^{j+1})$ where $v(a^{j+1})$ is a virtual attribute for the sequence Σ_{j+1}^m and a^{j+1} is either ϵ or an attribute of AG^{j+1} such that $a^{j+1} \in A(type(a^j))$.

At the beginning of a virtual construction sequence we suppose that we know a virtual attribute. We will see later that this virtual attribute is given by the last evaluator.

The coupling evaluator takes as input a virtual production instance, a virtual attribute and a position; the virtual attribute and the position give the virtual construction to be applied on the virtual production instance. The coupling evaluator gives as output the virtual attribute for the rest of the sequence and the output of the virtual construction, i.e. the new virtual production instance and the new position.

Definition 10 (Coupling Evaluator). Let $AG : G \rightarrow G'$ be an attribute coupling. The *Coupling Evaluator* of AG, noted CE, is a set of *virtual transitions* defined as follows:

1. for each production $p : X_0 \rightarrow X_1 \ldots X_n$,
2. for each attribute rule $f_{p,y^0,Y^0} : Y^0.y^0 = p'(Y^1.y^1, \ldots, Y^l.y^l)$ on p,
3. for each attribute occurrence $Y^{j'}.y^{j'} = X_j.a, j' \in [0..l]$, of this rule,

we create the virtual transition $< V(p), v(a), j > \Longrightarrow < V(p'), v(a'), j' >$ where:

$V(p)$ is a virtual production instance for some sequence of attribute couplings ending with AG,

[3] for convenience we define that $a.\epsilon = a$.

$v(a)$ is a virtual attribute for some sequence of attribute grammars starting with AG,
j is a position in p,

and such that:

- $v(a) = a.v(a')$, and
- $< V(p), j > \underset{a}{\Longrightarrow} < V(p'), j' >$.

This definition is completely independent of the sequence Σ in which this attribute coupling is used. So we reach our aim to have a separately constructible evaluator.

The position j in the virtual transition represents the position of the calling virtual node in the current virtual production instance. This position can change at each step.

Fig. 3 presents the coupling evaluator for our running example of AC.

4 Parameterizable Evaluator

The type of parameterizable evaluator is not relevant to our construction, i.e., these parameterizable evaluators can be based on different evaluation methods. So, for simplification, we do not give a formal definition of such an evaluator and we limit our presentation to the use of evaluators based on visit sequences (see figure 4). A parameterizable evaluator is the part of a classical evaluator where attribute computations are made. It takes information about its input tree through parameters.

In the different definitions, for the sake of simplicity, we did not introduce the visit numbers, but it is easy to introduce them, passing the visit number through the different coupling evaluators.

5 Evaluation Strategy

Now, to complete our construction of a sequence of coupling evaluators, we must define the *visit selector*, which finds the real production in the input syntax tree and calls the first coupling evaluator on this production.

The virtual node on which the current visit is called gives the real node of the current call and the virtual attribute which will direct the subsequent sequence of virtual constructions.

For convenience, we assume that we know the father node, noted u^f, of the current production instance. It allows us to find the new production instance and the position of the call node in it.

Definition 11 (Visit Selector). Given a parameterizable evaluator based on visit sequences for the AG which ends a sequence Σ_1^m, we define a *Visit Selector*, noted VS, such that:

For a given tuple of form $< V(p), v(a), j >$
the coupling evaluator has the following form:

case p is

$p_1 : Z \to L$ $\quad < V(p) = V(p_1) : v_0 \to v_1, v(a) = a.v(a'), j >$
case a.j is
$z.0 :< V(p_1) : v_0.z \to v_1.s, v(a'), 0 >$
$s.1 :< V(p_1) : v_0.s \to v_1.s, v(a'), 1 >$
$h.1 :< V(p_3) : v_1.h \to, v(a'), 0 >$

$p_2 : L_0 \to L_1 D$ $\quad < V(p) = V(p_2) : v_0 \to v_1 D, v(a) = a.v(a'), j >$
case a.j is
$s.0 :< V(Id_L) : v_0.s \to v_1.s, v(a'), 0 >$
$s.1 :< V(Id_L) : v_0.s \to v_1.s, v(a'), 1 >$
$h.1 :< V(p_2) : v_1.h \to v_0.hD, v(a'), 0 >$
$h.0 :< V(p_2) : v_1.h \to v_0.hD, v(a'), 1 >$

$p_3 : L \to$ $\quad < V(p) = V(p_3) : v \to, v(a) = a.v(a'), j >$
case a.j is
$s.0 :< V(Id_L) : v.s \to v.h, v(a'), 0 >$
$h.0 :< V(Id_L) : v.s \to v.h, v(a'), 1 >$

$Id_L : L_0 \longrightarrow L_1$ $\quad < V(p) = V(Id_L) : v_0 \to v_1, v(a) = a.v(a'), j >$
case a.j is
$s.0 :< V(Id_L) : v_0.s \to v_1.s, v(a'), 0 >$
$s.1 :< V(Id_L) : v_0.s \to v_1.s, v(a'), 1 >$
$h.1 :< V(Id_L) : v_1.h \to v_0.h, v(a'), 0 >$
$h.0 :< V(Id_L) : v_1.h \to v_0.h, v(a'), 1 >$

Fig. 3. Coupling Evaluator for the List Inversion AG

- given a calling virtual node $v = u.a^1. \cdots .a^{m-1}$ for the sequence Σ,
- given the father node u^f of the current production instance of G^1,

we build a starting tuple for the sequence, noted $< V(p), v(a), j >$, and the new u^f where:

1. $v(a) = a^1. \cdots .a^{m-1}$;
2. if the real calling node u is equal to u^f then we go up in the input tree, $u^f \leftarrow father(u)$, and the calling node position j in this new production is such that $u = Pos(u^f, j)$; otherwise, i.e. if u is different from u^f, then we go down in the input tree, $u^f \leftarrow u$, and the calling node position j in this new production is 0;
3. $V(p) : u^f \to Pos(u^f, 1) \cdots Pos(u^f, n)$ with $p = prod(u^f)$.

Starting State Given a sequence of attribute couplings Σ_1^m, we now have all the elements to construct the whole evaluator; we just need to define the starting

The evaluator of List Inversion based on visit sequences is parameterized by the tuple $< V(p), \epsilon, 0 >$. The calling position is always equal to zero because we use a top-down evaluator. This evaluator does not perform multiple visits to a same node, so we do not introduce visit numbers.

$$
\begin{array}{lll}
\textit{case } p \textit{ is} & & \\
& p_1 : Z \rightarrow L & < V(p) = V(p_1) : v_0 \rightarrow v_1, \epsilon, 0 > \\
& & v_1.h := p_3(); \\
& & \textit{visit } v_1; \\
& & v_0.z := p_1(v_1.s); \\
& & \\
& p_2 : L_0 \rightarrow L_1 D & < V(p) = V(p_2) : v_0 \rightarrow v_1 D, \epsilon, 0 > \\
& & v_1.h := p_2(v_0.h, D); \\
& & \textit{visit } v_1; \\
& & v_0.s := Id_L(v_1.s); \\
& & \\
& p_3 : L \rightarrow & < V(p) = V(p_3) : v \rightarrow, \epsilon, 0 > \\
& & v_0.s := Id_L(v_0.h); \\
& & \\
& Id_L : L_0 \longrightarrow L_1 & < V(p) = V(Id_L) : v_0 \rightarrow v_1, \epsilon, 0 > \\
& & v_0.s := Id_L(v_1.s); \\
& & \textit{visit } v_1; \\
& & v_1.h := Id_L(v_0.h);
\end{array}
$$

Fig. 4. Parameterizable Evaluator for the List Inversion AG

state. At the beginning of the process, the first tuple $< V(p), v(a), j >$ is defined as follows:

1. $V(p)$ is the root production of the input tree;
2. $v(a)$ is composed of the sequence of the synthesized attributes of the start symbol (root) of each attribute coupling except the last one, $z^1.z^2.\cdots.z^{m-1}$;
3. j is equal to 0.

The starting current node u^f is the root node.

Evaluator Chaining Given a sequence Σ_1^m, the call of the parameterizable evaluator PE^m of AG^m on the input syntax tree of G^1 is made using the different coupling evaluators CE^i. The call of the new visit sequence by PE^m is passed to CE^1 using the visit selector VS. This process is illustrated in Fig. 5. The links between these different entities lead to an evaluation method which can walk up and down in the input syntax tree.

The status of the last evaluator seems to conflict with the idea of separate compilation (an AC by itself should not know that it is the last in some sequence). This is not true if we consider that, for each AC, we have to construct both a coupling evaluator and a parametrizable evaluator. We claim that this will be

The call on the virtual node $v = u.v(a^1)$ is performed as follows:

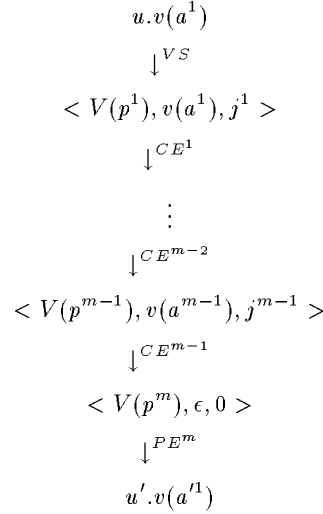

Fig. 5. Example of evaluation sequence

the case in practice because, in the course of the development of a whole AC sequence, you have to test each AC in turn and hence have to construct its parametrizable evaluator.

6 Extended Coupling Evaluators

In this section, we give a brief idea of how to introduce semantic computations in an AC while keeping our notion of coupling evaluator. We also briefly present how to introduce complex tree-construction rules, such as conditionals.

We have presented the coupling evaluator only for purely syntactic ACs, in which each set of attribute rules is composed only of (simple) tree-construction rules. We would like to accept a sequence of ACs with some semantic rules. First we suppose that we have a mechanism able to separate each AC into two AGs. In [6] we can find such a decomposition, called *e-t-Decomposition*. The first one only contains semantic rules and is called the semantic AG; the second one, called the syntactic AG, is purely syntactic and contains only tree-construction rules. Our construction can be applied to the syntactic AG.

For these two AGs, we construct their evaluators, a parameterizable evaluator for the semantic AG, and a coupling evaluator for the syntactic AG. Then the introduction of the semantic part is reduced to correctly calling the parameterizable evaluator of the semantic AG before any call to the coupling evaluator of the syntactic AG. To do so, it is necessary to correctly store some attribute instance values of semantic AGs at the real nodes of the real input tree. Indeed,

several calls of the same coupling evaluator can use a same attribute instance value in different contexts but on the same input tree. However, it must be clear that each semantic attribute of each AC is directly attached to the real node corresponding to the virtual node that carries it; no additional data structure is needed.

To accept complex tree-construction rules, such as conditionals, we introduce into the coupling evaluators complex expressions of the same form to regroup virtual transitions. Then the idea is simple: the complex tree-construction rules are decomposed into simple tree-construction rules, to which our virtual transition construction can be applied; then we recombine the results into a complex virtual transition using the previously introduced complex expressions. Details can be found in [13].

We have also eliminated many useless operations introduced by identity rules creating non-local virtual production instances. The details of this elimination can be found in [12].

7 Related work

In [6], there exists a preliminary approach which constructs directly from the evaluators of an AG sequence, an evaluator associated to the AG resulting from the descriptional composition of the sequence. But this approach only applies to purely synthesized syntactic AGs[4]. More precisely, "directly" means that there is no need to compute the attribute dependencies for the composed AGs; however the need to completely construct the evaluator still exists. Our approach has some similarities with this construction, except that we work directly on the evaluators and we accept non-purely-synthesized AGs.

In [3], the authors present the notion of separable AGs. They want to have separate compilation of these AGs, hence they do not use descriptional composition. They propose an adaptation of the classical evaluation scheme which allows separate compilation, but the construction of (some) intermediate trees is necessary. The aim of our approach is to allow separate compilation while avoiding the construction of any intermediate tree. From the external point of view, then, both works appear to have similar expressive power (separable AGs) but it is from an evaluation point of view that our approach is very different, essentially because we do not construct the intermediate trees.

8 Conclusion and Future Work

The main advantage of the notion of coupling evaluator is certainly that there is no class constraint on the coupling evaluators. The class of the resulting (complete) evaluator is the class of the parameterizable evaluator of the last AG of a given sequence Σ. Thus, to construct an evaluator based on visit sequences, for a given AC sequence, only the last AG needs to have such an evaluator.

[4] See proposition 8 in [6] for more details.

In fact, the resolution of class constraints in descriptional composition was the first motivation of this work. In the same way, it is possible to see our coupling evaluator as a generalization of the classical AG transformations for class reduction (e.g. from SNC to l-ordered [2]; see also [1]) which often work at the specification level.

The second advantage is the possibility to construct the coupling evaluators separately while keeping the good properties of descriptional composition, i.e. the fact that the intermediate trees are not constructed. However, only practical experience with a real implementation (which we don't have yet) will show whether this is indeed valuable.

Regarding the size of the data structures needed for our Coupling Evaluator, we notice that it is equivalent to the size of the data structures which is needed for descriptional composition. Since our Coupling Evaluators only work on the part of a grammar which is really necessary, the size of their data structure is smaller than the one needed for a sequence of classical evaluators which construct the whole intermediate trees.

We must notice that, if we are not interested in separate compilation, we can obtain much more efficient combined evaluators. Indeed, if we know the sequence of ACs, we can directly code the different tests into the coupling evaluators, because the number of cases which are really used is very small. This is a form of specialization by partial evaluation. This also reduces the size of the evaluator. More precisely, it is possible to apply a transformation on a given sequence of Coupling Evaluators which produces a particular evaluator where only revelant cases are present and where a sequence of calls of coupling evaluators is reduced to a unique call. This new evaluator retains the advantage of the descriptional composition and it is also always possible to construct it.

Moreover, we presented a first approach to the construction of Coupling Evaluators, where the main goal was not the efficiency of this kind of evaluator, but rather "simplicity".

In this paper we have assumed that each attribute occurrence was used only once in each production of the attribute coupling. We think that this restriction can be relaxed rather simply, by remembering which attribute occurrence has been chosen until a visit returns.

We also hope to eliminate many useless operations introduced by identity rules, by creating non-local virtual production instances and trying to find statically useless copy rules; this analysis will apply the results and techniques of [12] to descriptional composition.

From a theoretical point of view, it is very difficult to formally compare sequential implementation with this new technique, beside the fact that intermediate trees are not to be constructed. The only good (but partly false) view which we could give, is a comparison with factoring in mathematics: with coupling evaluators we factor out moves into one tree (see Fig. 6).

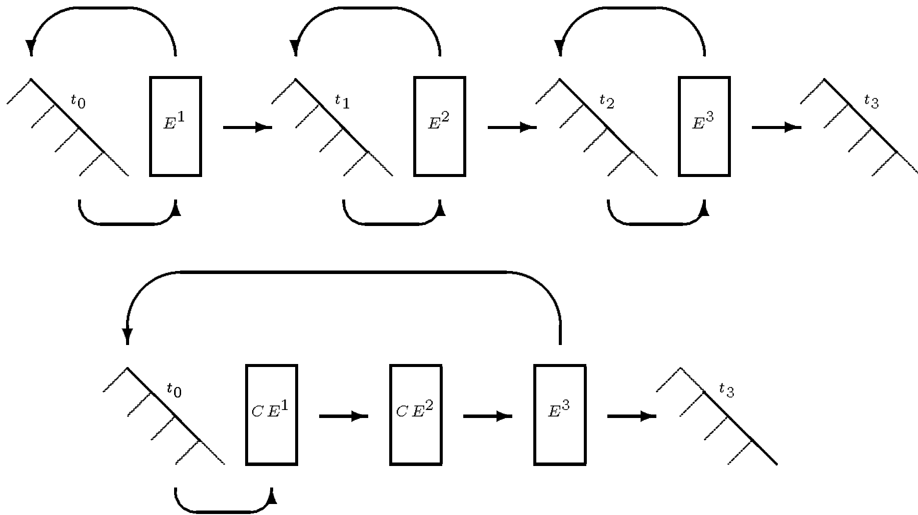

Fig. 6. Factorization of moves

References

1. Deransart, P., Jourdan, M. and Lorho, B. *Attribute Grammars: Definitions, Systems and Bibliography*. Lect. Notes in Comp. Sci., vol. 323, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.
2. Engelfriet, J. and Filè, G. Simple Multi-Visit Attribute Grammars. *J. Comput. System Sci. 24*, 3 (June 1982), 283–314.
3. Farrow, R., Marlowe, T. J. and Yellin, D. M. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *19th ACM Symp. on Principles of Progr. Languages* (Albuquerque, NM, Jan. 1992). pp. 223–234.
4. Ganzinger, H. and Giegerich, R. Attribute Coupled Grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction* (Montréal, June 1984). *ACM SIGPLAN Notices 19*, 6 (June 1984), 157–170.
5. Ganzinger, H., Giegerich, R. and Vach, M. MARVIN: a Tool for Applicative and Modular Compiler Specifications. Forschungsbericht 220, Fachbereich Informatik, Univ. Dortmund, July 1986.
6. Giegerich, R. Composition and Evaluation of Attribute Coupled Grammars. *Acta Inform. 25* (1988), 355–423.
7. Jourdan, M., Le Bellec, C., Parigot, D. and Roussel, G. Specification and Implementation of Grammar Couplings using Attribute Grammars. In *Programming Languages Implementation and Logic Programming* (Tallinn, Aug. 1993), M. Bruynooghe and J. Penjam, Eds. Lect. Notes in Comp. Sci., vol. 714, Springer-Verlag, New York–Heidelberg–Berlin, pp. 123–136.

8. Jourdan, M. and Parigot, D. Application Development with the FNC-2 Attribute Grammar System. In *Compiler Compilers '90* (Schwerin, Oct. 1990), D. Hammer, Ed. Lect. Notes in Comp. Sci., vol. 477, Springer-Verlag, New York–Heidelberg–Berlin, pp. 11–25.
9. Jourdan, M., Parigot, D., Julié, C., Le Bellec, C. and Durin, O. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *ACM SIGPLAN '90 Conf. on Progr. Languages Design and Implementation* (White Plains, NY, July 1990). ACM SIGPLAN Notices, vol. 25, no. 6, pp. 209–222.
10. Knuth, D. E. Semantics of Context-free Languages. *Math. Systems Theory 2*, 2 (June 1968), 127–145. Correction: *Math. Systems Theory 5*, 1, pp. 95-96 (Mar. 1971)..
11. Le Bellec, C. La généricité et les grammaires attribuées. Thèse de doctorat, Dépt. d'Informatique, Univ. d'Orléans, June 1993.
12. Roussel, G. A Transformation of Attribute Grammars for Eliminating Useless Copy Rules. Research report to appear, INRIA, 1994.
13. Roussel, G. Différentes transformations de grammaires attribuées. Thèse de doctorat, Dépt. d'Informatique, Univ. de Paris VI, Mar. 1994.

This article was processed using the LaTeX macro package with LLNCS style